Evolving Extensional Representations of Concerns

Sukanya Ratanotayanon
Dept. of Informatics
University of California, Irvine
+1 949 824 2373
sratanot@uci.edu

Susan Elliott Sim
Dept. of Informatics
University of California, Irvine
+1 949 824 2373
sesim@uci.edu

Derek J. Raycraft
Dept. of Informatics
University of California, Irvine
+1 949 824 4047
draycraf@uci.edu

Abstract

There is a growing body of work using concerns to aid developers when evolving software systems. These systems typically rely on concern maps to represent concerns. A concern map pulls together the different sections of the code implementing a concern, thus mitigating the difficulties in working with scattered and crosscutting concerns. However, there has been comparatively little work on evolving these concern maps along with the underlying software. Work to date has focused on evolving intensional representations of concerns. In this paper, we present an approach to storing extensional representations of concerns and evolving them over time. Our approach uses the history from a revision control system to maintain the accuracy of the concern maps. We have developed, Zelda, an Eclipse plug-in for creating and maintaining extensional representation of concerns. Zelda's ability to maintain links was evaluated in an empirical study on releases of jEdit, an open-source text editor written in Java (260 KLOC). Evolution of links was performed for five concern maps. The results showed that, for the entire set of 419 links across the five concern maps, after 25 versions the precision was 0.90 and the recall was 0.73. The average precision and recall per concern map is 0.78 and 0.69 respectively.

Categories and Subject Descriptors D.2.3 **[Software Engineering]**: Coding Tools and Techniques – Programmer editors. D.2.6 **[Software Engineering]**: Programming Environments – Programmer workbench.

General Terms Documentation, Experimentation.

Keywords Concern Evolution, Extensional Representations, Concern Maps, Link Maintenance

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

1. Introduction

Cross-cutting concerns make software evolution challenging, because relevant sections of code can be spread across many classes and documents [1]. Therefore, being able to represent these concerns can help subsequent software developers to find relevant information more quickly. Concerns can be represented intensionally or extensionally [2]. Intensional representations use rules to characterize participants in the concern map, whereas extensional representations enumerate the participants explicitly. Often intensional rules (e.g. callers of a method) can be used to generate an extensional representation (e.g. a list of callers). While there has been work creating both types of representations, there has been little work on maintaining them as the software evolves. However, these concern maps need to be kept up-to-date to ensure their usefulness and to encourage developers to create them.

One of the most challenging issues in representing concerns extensionally is maintaining the representation when underlying artifacts are changed. This is particularly a problem in fast-paced environments using incremental software development. The representation can deteriorate quickly as software evolves, unless significant effort is expended to keep concern maps up to date. Even short-lived concerns need to be maintained, because the information might be useful later, for example, as group memory. We need to be able to present the user with accurate links even after they are affected by changes.

There have been two studies by Robillard [3, 4] on maintaining and evolving intentional concerns, and little work on maintaining extensional concerns. In this paper, we present an approach to storing extensional representations of concerns and evolving them over successive changes. Our approach uses the history from a revision control system to maintain the accuracy of the concern maps. We have developed, Zelda, an Eclipse plug-

in for creating and evolving extensional representation of concerns.

In order to evaluate the effectiveness of our approach in maintaining the representation over successive changes, we performed an empirical study with jEdit an open-source text editor written in Java (260 KLOC). The study was done by tracing the evolution of five concern maps for 25 releases after their creation. This period includes over 2,000 incremental revisions. The concerns maps were created based on changes made in selected commit transactions occurring before jEdit 4.1 pre 5. The resulted links in the concern maps lead to both source code and non-code text files such as documentation and configuration files. The result showed that, for the entire set of 419 links across the five concern maps, after 25 versions the precision was 0.90 and the recall was 0.73. The average precision and recall per concern map is 0.78 and 0.69 respectively.

The remainder of the paper is organized as follows. Section 2 reviews previous work on concerns. Section 3 presents a discussion about extensional and intensional concern representations. Section 4 presents our approach to create and maintain extensional concern representation. We discuss our approach in evolving the extensional concern representations in Section 5. The Zelda tool, which implements our approach, is described in Section 6. The evaluation of Zelda is given is Section 7. Section 8 discusses future work, followed by concluding remarks in Section 9.

2. Background

A concern is any aspect of a software system that a stakeholder considers to be a coherent concept; this can be a feature, a requirement, or design pattern [3]. A concern graph or map is a mapping of concerns onto portions of source code. These concern maps can be described intensionally or extensionally [2, 4]. An intensional description is a rule describing participants in the map, such as all accessors of a variable in a particular class. An extensional description is an explicit enumeration of the participants in the map. Typically, an intensional description is used to generate an extensional list.

To document cross-cutting concerns, many tools also allow developers to create models of concerns. Most tools represent concerns extensionally, including ConcernMapper [5], and a concern toolset in INARI [6]. However, some tools permit concerns to be represented intensionally and extensionally, such as FEAT and CME [7].

Evolving concerns maps as the underlying software artifacts change is a critical problem. Developers are less likely to put effort into creating concern maps if they become out of date quickly. While there has been significant work on finding concern maps, either by analyzing software artifacts [3, 8]or by monitoring developers as they work [9-11], there has been relatively little work on evolving concerns maps.

Robillard has previously studied how Markers concern in the JEdit, an open source advanced editor, evolved over time [12]. The markers concern consisted of 11 extensional fragments and 24 intensional fragments, involving 60 participants in 11 different classes. He found that over 32 versions of JEdit, inconsistencies appeared in the concern graph in 10 versions. The number of inconsistent fragments ranged from 2 to 21—more than 60% of the fragments in the worst case. In another study, the Syntax Highlighting concern was tracked over 34 versions of JEdit [4]. Initially, a concern map was created manually with 24 participants. Using only an extensional representation, 17 of the participants became inconsistent. However, the addition of a tool, ISIS4J, helps infer in new concern participants based on structural properties of the concern map. Reinikainen et al. have studied evolving extensional concern maps by checking identifiers in new code [6].

Despite the greater prevalence of tools using extensional representations, there are fewer techniques for evolving them. Our contribution falls into this space: We present an approach to automating the management and evolution of extensional representations of concerns when their underlying artifacts change. Our approach is complementary to the techniques for evolving intensional representations discussed above, and both are necessary to evolve concern maps.

3. Intensional Versus Extensional Concerns

In the most general terms, a concern is anything of interest or importance to a stakeholder in a software product [13]. They can arise at any stage in software development. Examples of concerns are features, non-functional requirements, and domain-independent units such as logging. Modern software systems are highly complex, which often results in concerns that cannot be easily encapsulated. Furthermore, the concerns are amorphous, which means that cross-cutting concerns are inevitable. A cross-cutting concern results in a delocalized plan [14], because a conceptual unit has its implementation scattered in multiple locations. Cross-cutting concerns increase the difficulty of program comprehension, because they are by nature tangled and scattered. This problem can make it more difficult to evolve software. Therefore, an ability to map a concern to its manifestations in software implementation is highly valuable.

3.1 Definitions

In order to discuss the differences between intensional and extensional representations of concerns, we need to establish some definitions.

A binary relation, $R=(X, Y, G)$, is a mapping between two sets (or classes). X and Y are arbitrary sets, and G is a subset of the Cartesian product $X \times Y$. The sets X and Y are called the domain and codomain, respectively, of the relation, and G is called its graph [15].

A concern map is a binary relation, such that X is a set of all concerns and Y is a set of all atomic elements. By atomic elements, we mean those can be uniquely identified in the universe of discourse, i.e. the text files, source code, or program units. The Cartesian product $X \times Y$ is the set of all possible links between concerns and atomic elements. The graph G is this context is a specific set of these links and corresponds to a concern map.

There are two ways to represent a concern map, extensionally or intensionally. An intensional representation of G uses a rule, operation, attribute, or pattern to specify members of the graph. For example, if a concern c_1 consists of all the methods that read variable v_1, then R the concern map can be expressed as the following.

X= {All Concerns}
Y= {All atomic elements}
$v_1 \in V$ where V= {All variables}, $V \subseteq Y$
$M = \{All\ methods\}$, $M \subseteq Y$
Let G={x,y| x $\in \{c_1\}$, $y \in \{\forall m \in M | m$ uses $v_1\}\}$

A concern map can also be represented extensionally by explicitly specifying or listing elements that are members of the concern. An extensional representation of G relies on a list or enumeration of members from $X \times Y$. A possible extensional representation of the concern map given above, could be the following.

X= {All Concerns}
Y= {All atomic elements}
$v_1 \in V$ where V= {All variables}, $V \subseteq Y$
$M = \{All\ methods\}$, $M \subseteq Y$
$\{m_1, m_2, m_3\} \subseteq M$
Let G= $((c_1, m_1), (c_1, m_2), (c_1, m_3))$

These two concern maps would be equivalent if there were exactly three methods that read variable v_1.

These distinctions in specifying the concern map lead to differences in capabilities for representing concerns.

3.2 Essential Differences

Intentional and extensional representations of concerns both have their place in program navigation tools. The choice of which to use is guided by the characteristics listed in Table 1.

The first characteristic is how each type of representation is specified. As mentioned above, intensional representations are rule-based, while extensional representations are primarily enumerations. Extensional representations are simple; one needs only to be able to identify set members. Furthermore, they are portable, meaning that sets produced by different tools can be combined easily. On the other hand, intensional representations can be more difficult to use because rules can be complex and finding an appropriate rule can be challenging

Table 1: Differences between Concern Representations

	Intensional	**Extensional**
Specification	• Rule-based, using properties, principles, or patterns. • Complex	• Explicit list of participants or members • Simple • Portable
Size of *G*	• Unbounded	• Bounded
Evolvability	• High • Participants updated by re-applying the rules.	• Low • Each participant must be updated individually
Limitations	• Rules limited by available grammar • Rules can be difficult to write. • Potential mismatch between expressed rule and intention. • Harder to combine results from different tools.	• Effort required to maintain. • Not amenable to computation.

The size of the graph G of each type of representation, i.e. the number of possible set members from X × Y that are included in the specification, also differs. A rule that specifies an intensional representation has a potentially unbounded number of participants. In other words, this number is not fixed when the specification is written. In contrast, the number of participants in an extensional representation is given and bounded.

Intensional representations are relatively easy to maintain as the underlying software evolves. The rules or patterns can be applied to new versions of the code and the referents identified using a grammar. Any new code that matches the rules is added automatically. By the same token, code that no longer matches is removed automatically. However, set members are lost when the

domain changes, i.e. the variable in the rule cannot take on the desired value because the appropriate set member is no longer available. This situation may occur, for example, when a program element's identifier changes.

On the other hand, extensional representations are much harder to maintain. Each member of G must be updated individually when the code is changed. Adding and removing participants also requires effort. In this situation, set member can be lost when the codomain changes. In both cases, concern map participants can be lost to name changes and large-scale changes, such as refactoring.

Based on these three characteristics, each representation has its limitations. For intensional representations, the kinds of concerns that can be specified are limited by the grammar of the rules. Rules expressing exactly the concern desired can be difficult to write. For instance, a developer may specify a concern using a rule that is easy to write rather than using a more complex rule that expresses her intention more accurately. It can be difficult to combine binary relations, even when the domain and codomain are the same, because the rules may be conflicting or contradictory.

Extensional representations have two main limitations. When the code changes, every member of the binary relation needs to be checked and updated, which can entail a great deal of effort. The second limitation is that a list of set members is less amenable to computation reasoning or algebra. As a result, it may be more difficult to check properties or verify extensional representations of concerns.

3.3 Accidental Differences

One characteristic that is common to both intensional and extensional representations of concerns is that neither is restricted to a particular type of atomic elements that can be denoted as part of Y. Restrictions only comes about as a result of design decision in their implementation. Either one can be used to refer to program elements or text elements. The granularity of program elements can be the class, method, and block. The granularity of text units can be files, lines, words, or characters. For instance, the atomic elements in FEAT, ConcernMapper, and INARI are program elements, such as methods, variables, and classes. In contrast, the atomic elements in our tool are lines in text files, including source code.

4. Modeling Extensional Concern Maps

In the previous section, we established that concern maps can be thought of as binary relations of the form R=(X, Y, G), where X is the set of all concerns, Y is the set of all atomic elements in the software project, and G is a graph that relates the two.

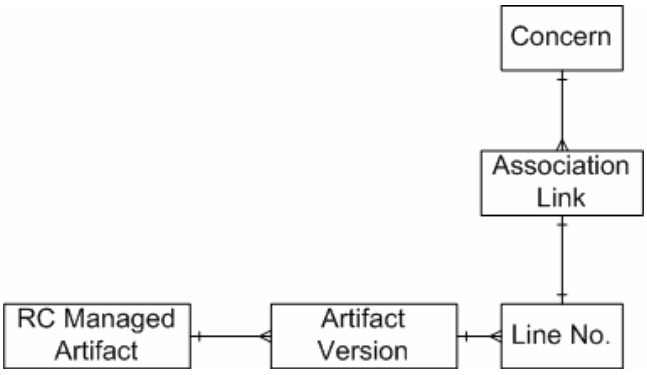

Figure 1. Model of the Concern Map.

Since we wish to support extensional representations of concerns, we want to be able to link to an arbitrary work product at a relatively small level of granularity. To this end, we made the atomic units lines in text files. This allows the representation to work with a source code from a variety of programming languages, non-code text such as README notes and configuration files, and artifacts from different phases in the software lifecycle. Also, this decision allows us to leverage an important existing, widely used software tool, the revision control system. This allows us to track the evolution of the underlying software and to evolve the concern map along with it. Furthermore, it foregrounds the concept of evolution in our system design and we can associate a concern map with versions of the system.

Our model for concern maps based on our definitions is shown below in **Figure 1**. A Concern is a member of the set X. The bottom three boxes together denote a member of the set Y, that is, a particular line in a specific version of a file under revision control. It should be noted that these three boxes don't represent the line itself, but provide a way to access the line. The Association Link box represents a member of the set G, that is, an participant in the concern map.

One limitation of this model is that concerns are not placed under revision control. Therefore, we do not have a way to evolve them as well as the concern maps. This issue is beyond the scope of this paper where we focus on evolution of concern maps as the underlying artifacts change and has not been addressed by any concern management tool. Another limitation of our approach is we do not take advantage of the structure and information in the abstract syntax tree (AST). However, it would be a simple matter to add details obtained from other static analysis tools to the concern map, because we are using an extensional representation.

4.1 Constructing the Concern Map

Extensional representations of concerns involve an explicit list of locations or elements in the source code, which means that someone must specify this enumeration at some point. This activity can involve significant effort, especially when there are many participants and many concerns. Therefore, appropriate tool support for users is essential, because software developers rarely have the luxury of time or patience to spend on non-coding activities.

To this end, we incorporate the association mechanism into an integrated development environment (IDE), where links can be added to the concern map manually or automatically. To add links manually, a developer selects particular lines and adds them to the relevant concern. To add links automatically, the user can choose to have links created when changes are committed to the RC system. Both approaches can be used in combination to construct a concern map. In both cases, the links between the concern and the lines will be created and stored in the database.

5. Evolving The Extensional Representation

Our extensional representation of concerns consists of links pointing to lines in specific versions of artifacts. Consequently, the referents of these links can become inconsistent as the underlying artifacts are changed and would lead to incorrect concern maps. However, it is infeasible to expect developers to update these links manually. Therefore, we provide an approach to automatically update the location of links over successive changes by using information queried from an RC system. There are many benefits to representing concerns in this manner and leveraging the revision control system. Working in tandem with a revision control system that maintains the artifacts, it is always possible to go back to the original content of a link target or any other intermediate versions.

Our algorithm relies on the 'diff' file that summarizes the differences between two files. A tool produce the diff file is commonly included in the RC system, and is also a part of UNIX operating system. As depicted in **Figure 2**, we need to input into the comparison utility with the version of the file in which the links are created and the current version of the file in order to receive the diff result use for updating links.

An example of a diff result in the unified format can be seen in the box at the top right of **Figure 2**. It contains header information, such as the files that are compared, followed by one or more "change chunks." A change chunk begins with a header demarcated by the double ampersands (e.g. @@) that shows the range of the change in both files. The markers bracket a description of the

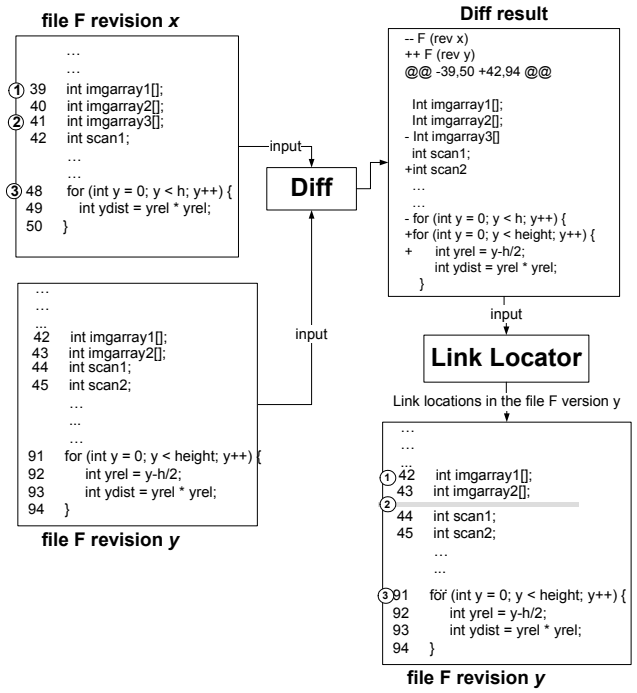

Figure 2. Determine link endpoint locations with diff result

differences between two files. Symbols at the beginning of the line indicate whether a line is found in only one file or the other, or both. Using the example diff result in **Figure 2**, a line starting with '-' is the line that only exists in the version *x* and a line starting with '+' is the line that only exists in the version *y* of the file. A line that has no starting symbol exists in both files and signifies that no changes have been made. With this information provided by diff file, a map between lines in two different versions of a file F_x and F_y, can be created and used to determine the updated location of the link. For example, line number 39 in F_x can be mapped to line number 42 in F_y. Therefore, when presenting concern map in F_y, the link 1 should point to line 42 instead of line 39, which was recorded originally.

The updating of links can have three possible states as shown in **Figure 2**. An "unchanged" line has the same content, but may have moved within the file hence have different locations. For instance, the line linked to by link 1. The link 2 is an example of a "removed" link cannot be found, and is not shown to the user. A "modified" link is one where the content has changed, and may have moved. These links will be presented, but with a different appearance than the 'unchanged' ones. See the link number 3 for an example of an unchanged link.

5.1 Algorithm

Based on the description above, the algorithm for evolving the extensional concern representation is given here. The algorithm takes as input two versions of a file, a file containing the diff in a unified format between the two

Let F_x represent version x of file F
Let F_y represent version y of file F, where y > x
Let d_{xy} be the diff result between F_x and F_y

Let l_x be the location of the link in version x
Let l_y be the location of the link in version y

We need to find l_y

Case 1: l_x does not appear in d_{xy}

Find the change chunk in d_{xy} that is closest to l_x

If (l_x occurs before the change chunk)
    Calculate distance d between the l_x and the first line of the change chunk in F_x
    Set l_y = first line of the change chunk in F_y - d

If (lx occurs after the change chunk)
    Calculate the distance d between lx and the last line of the change chunk in F_x
    Set l_y = last line of the change chunk in F_y + d

Case 2: l_x appears in d_{xy}

Find the change chunk in d_{xy} containing the l_x

Set p_x to the first line of the change chunk in F_x
Set p_y to the first line of the change chunk in F_y

Set $CC_{current}$ to the current line in the change chunk
Set CC_{next} to the line after $CC_{current}$

// Walk the two files
Iterate through the lines in the change chunk in the d_{xy}, updating both line pointers p_x and p_y, until $p_x == l_x$
    If $CC_{current}$ occurs in F_x and F_y, increment p_x and p_y by one
    If $CC_{current}$ occurs only in the F_x, increment p_x by one
    If $CC_{current}$ occurs only in the F_y, increment p_y by one

// Calculate the new location
If ($CC_{current}$ and CC_{next} occur only in F_x)
    then l_y = null
If ($CC_{current}$ occurs only in F_x and CC_{next} occurs in F_x and F_y)
    then l_y = null
If($CC_{current}$ occurs only in F_x and CC_{next} occurs in F_y)
    then $l_y = p_y + 1$
In all other cases, $l_y = p_y$

Figure 3. Link Evolution Algorithm.

files, and a link location in the older version of the file. The link evolution algorithm (depicted as the Link Locator box in **Figure 2**) computes the location of the link in the newer version of the file.

The algorithm breaks down into two cases. In Case 1, the link location of interest does not appear in the diff file. Consequently, the new location of the link is calculated by finding the distance to the closest change chunk. In Case 2, the location of interest does appear in the diff file. Two pointers are initialized to walk the two files based on the contents of the diff file until the link location of interest is found in the older file. By looking ahead at the next line, the new link location can be computed.

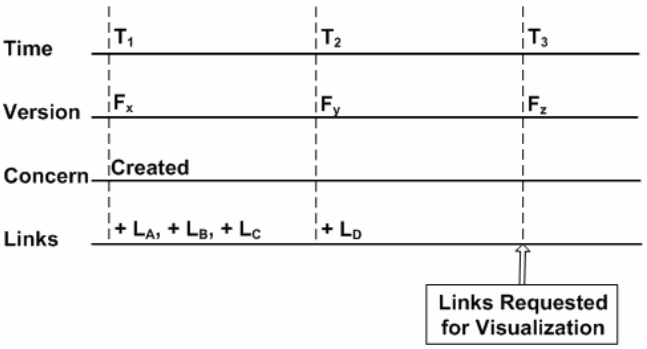

Figure 4. Time line of link creation.

It should be noted that our implementation of this algorithm only reads each diff file only once and constructs a map between two versions of a document. This approach is more efficient than locating each link in the new file one at a time.

5.2 Aggregating Representations Across Versions

Links in a concern map can be created at any time in the life of a project. Since our approach relies on comparing the version of the file where the link was added and the current version, we need to have a way to use the diff file between different versions paired with the current one. To illustrate this situation, let's consider the example depicted in **Figure 4**. In this situation, a concern contains links L_A, L_B, and L_C created in F_x at time T_1. Subsequently, at time T_2 in version y of the same file link L_D is added. When a developer chooses to view the concern map of this concern in F_z, the file in version z, we need to aggregate these links and retrieve their current positions in F_z.

The solution is to build a map of the lines between every file version where a link was injected and the current version. In other words, in order to situate links L_A, L_B, and L_C in F_z of the file, we use d_{xz}, the diff result between F_x and F_z. In order to locate link L_D in F_z, we use d_{yz}.

6. Zelda

The algorithm has been implemented in our tool, Zelda, an Eclipse plug-in. Zelda is a prototype platform for studying evolution of extensional representations of concerns. The front-end components are built as Eclipse views and editors. Back-end components include Subversion, a MySQL database, and concern management systems. The middle layer that manages the concern representations is also implemented under Eclipse.

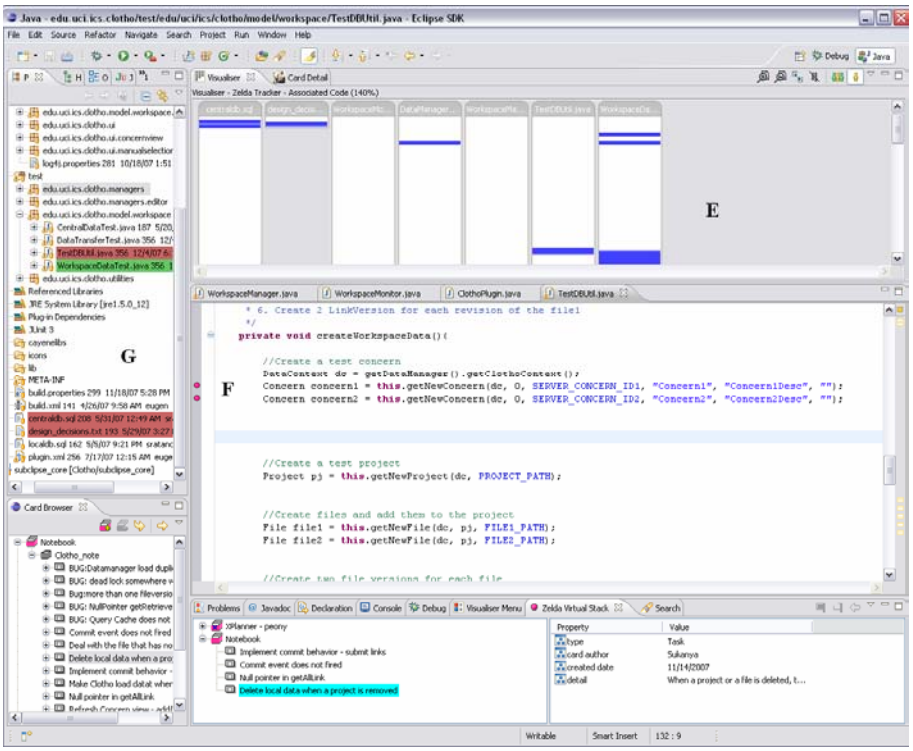

Figure 5. Zelda's Screenshot

6.1 Architecture

The software architecture of Zelda is depicted in **Figure 6**. In the back-end, Concern Providers manage information about concern that can be linked to associated lines. Examples of external Concern Providers are a bug tracker and an Extreme Programming (XP) management tool. We included an internal Concern Provider, called ConcernNotebook. The contents of ConcernNotebook are stored in a MySQL database. The information of the concern is retrieved and managed by Concern Manager when requested by the front-end components such as the Concern Browser. To allow us more flexibility in working with various card providers, differences among providers are shielded by the Concern Bus, which interfaces with different providers.

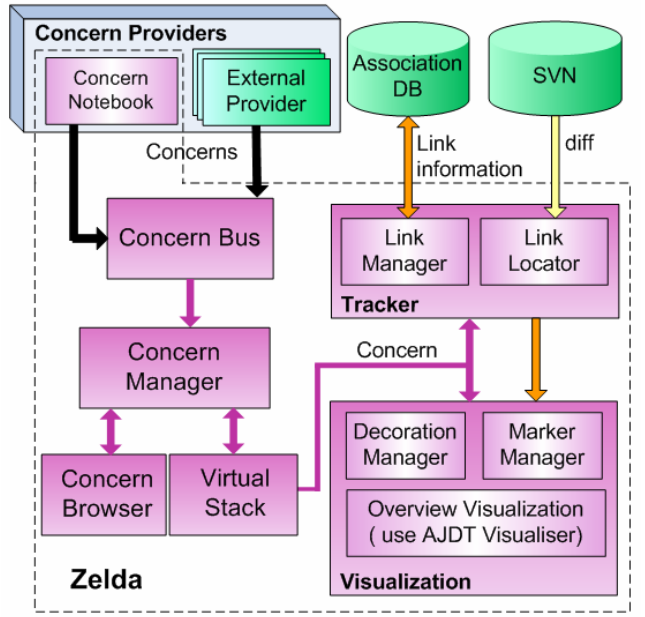

Figure 6. Zelda's Architecture

The front-end components allow users to interact with these concern maps. The Concern Browser shows a tree-view of the developer's concerns, while the Virtual Stack shows the current working set of concerns. The Tracker creates and manages link information. Associations can be created manually or automatically, as discussion in Section 4.1 . When links are retrieved, the Tracker uses the algorithm presented in Section 5.1 to update the location that they point to. The updated location of links will be presented by the Visualization components.

6.2 User Interface

A screenshot of the tool is shown in **Figure 5**. A developer can see a concern map of a specific concern using the following visualizations.

Overview: This visualization is labeled as E. We use the Visualiser plug-in from the AJDT group to implement this SeeSoft-style visualization [16] . It presents an overview of the concern map with associated files and associated lines

in the files. The files are shown as long blocks and the associated lines are shown as stripes within the blocks. The color encodes the status of the link whether it is modified or not. This overview provides easy access to the scattered fragments of the concerns. The developer can access a section of the concerns by double clicking on either the block or the stripe.

File Decorations: The participants of a concern map are presented at the file-level granularity using a file decoration. This visualization shows a different background on the items in the package explorer view in Eclipse, as shown in the area labeled G. By integrating the representation with the package explorer view, we can take advantage of the structural information of the software provided by the view.

Markers: Markers are used to show concern representation at the line-of-code level, as shown in the area labeled F. A custom marker is created and used to present the most recent location of each link. By using markers, links can be presented in a way that is grounded in the source code.

7. Evaluation

To evaluation our algorithm, we followed the evolution of five concerns over 30 releases of an open source software project. In this section, we first describe our experimental design, results, and threats to validity.

7.1 Experimental Design

For our study, we used jEdit [17], an open source advanced editor, as the subject system. We selected this software project because it has been used repeatedly on previous evaluations and so our results could be more easily compared with prior research. We worked with the distribution from version 4.1 pre1 (140 KLOC) to 4.3 pre 7 (260 KLOC), a total of 30 releases from May, 2002 to August, 2006, including 2,536 incremental revisions. We included not only the Java source code in our study, but also non-code text files, such as developer notes and documentation.

7.1.1 Selecting the Concerns

We created five concerns maps; each of them were constructed using the change set from a commit transaction. The criteria for selecting a concern are that the log message should be clear enough to tell the purpose of the change and the change needs to affect more than 3 files and touch different parts of the system. We started our selection process with version 4.1 pre1, because earlier versions had less detail in their revision control logs. We went through each commit transaction in order until we found five concerns that fit our criteria. Coincidentally, we used one change set from each version up to version 4.1 pre 5. To create a concern map for each of the selected concerns, we created links to lines affected by the change in the commit. The summary of these five concerns is shown in Table 2.

Table 2: Summary of Selected Concerns

Concern	No. of Files	No of Links
C_1: Auto indent bug fix	7	213
C_2: Folding bug fix	5	11
C_3: Find&replace bug fix	4	64
C_4: Adding new modes	4	10
C_5: NQC syntax highlight	13	121

The created concerns varied both in size (number of links) and the participants. All of them contained both code and non-code files. Also, all of our concerns contained links into the top ten most frequently changed files. These two characteristics allowed us to thoroughly exercise the concern map evolution algorithm in Zelda. The characteristic of the files linked to by these concern maps are presented in **Table 3**.

Table 3: Characteristic of Files Linked to by Concern Maps

Concern	Non-code File	Code File	Top-Ten Changed Files
C_1	2	5	4
C_2	3	2	2
C_3	1	3	1
C_4	3	1	1
C_5	3	10	3

7.1.2 Procedure

We used the following procedure to evaluate the correctness and completeness of the concern maps after each new release.

Given version v_i of a project and a concern map C_n created in that version, let l_{1i}, l_{2i}, l_{3i},… represent the location of the links. For each concern map, we performed the following procedure for every subsequent version of the project.

1. Calculate the new location of the links l_{1j}, l_{2j}, l_{3j},…in version v_j, where $j > i$.
2. Check the contents of the locations returned by Zelda.
    a. If no location was returned, the link is considered "removed" by the project developers.
    b. If the location returned contains a line that should be included in the concern map, the link is considered "correct."

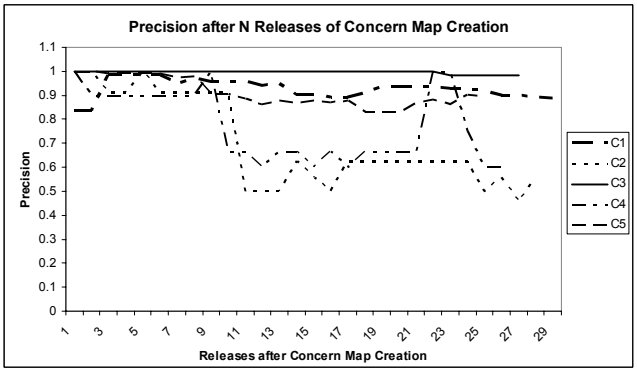

Figure 7. Precision at N Releases after Creation

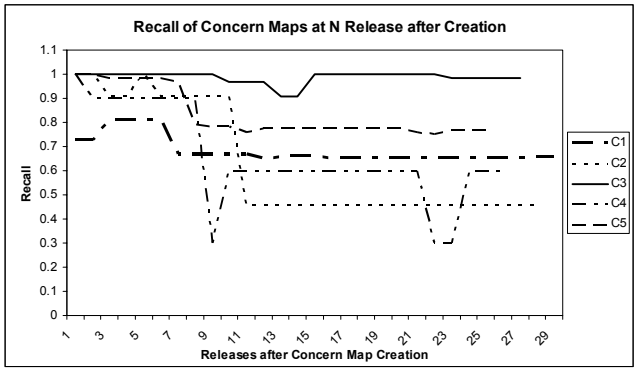

Figure 8. Recall at N Releases after Creation

c. If the location returned contains a line that should not be included in the concern map, the link is considered "incorrect."

3. Use these categorizations to calculate precision and recall for v_j, defined as follows.

Precision. The ratio of the number of relevant links retrieved per concern to the total number of links retrieved per concern.

$$\frac{\text{correct}}{\text{correct+incorrect}} \quad (1)$$

Recall. The ratio of the number of relevant links retrieved to the total number of relevant links.

$$\frac{\text{correct}}{\text{correct+incorrect+removed}} \quad (2)$$

We began this procedure for each concern map with the version immediately after it was created and continued for 25 versions. Consequently, we could compare the performance of the evolution algorithm for each concern after a given number of releases. We felt that this comparison was more fair than comparing them across a single release, because each concern would have undergone different numbers of versions.

7.2 Results

Overall, the algorithm performed well. For the entire set of 419 links across the five concerns, after 25 versions the precision was 0.90 and the recall was 0.73. The result is encouraging, because a large number of links were preserved during thousands of commit operations. The performance per concern was lower, with a precision of 0.78 and 0.69. The precision and recall of each concern map at this release are presented in Table 4. In the remainder of this section, we'll probe into the reasons and characteristics of this performance.

Small versus Large Concern Maps. We had two comparatively small concern maps, C_2, and C_4, with 11 and 10 links respectively. We also had two comparatively large concern maps C_1, and C_5, with 213 and 121 links respectively. We found that small concern maps had lower precision and recall, because the loss of one or two links would cause a 10-20% drop in performance. For example, the loss of 6 links for C_2 caused 44% drop in precision, but a loss of 18 links (three times as many) resulted in only an 11% decrease for C_1.

This tendency can be seen clearly in **Figure 7** and **Figure 8**. These graphs show the precision and recall of each concern map at *N* releases after creation. Losing a small number of links in C_2 and C_4 results in dramatic drops and jumps, while larger maps have more gradually changing rate.

Table 4: Results of Retrieved Links at the release 4.3 pre 7

Concern	Correct	Lost		Precision	Recall
		Removed	Incorrect		
C_1	139	62	12	0.89	0.66
C_2	5	1	5	0.56	0.45
C_3	59	0	1	0.98	0.98
C_4	6	0	4	0.60	0.60
C_5	93	17	11	0.89	0.76
Overall	302	79	33	0.90	0.73
Average per Concern				0.78	0.69

Precision versus Recall. We found that recall tended to be lower than precision, because links were more likely to be removed than retrieved incorrectly. Overall, 19% of the link participants were deleted, but only 9% were retrieved incorrectly. However, this is to be expected because during the course the software evolution, many lines or even files were removed due to the changes in features, bug repairs, or refactoring.

Increases in Precision and Recall Over Time. While we expected precision and recall to erode over successive versions, we found that they actually increased on occasion. We had expected that the precision and recall of

the concern maps will gradually decrease as more changes occur because the changes would cause the link endpoints to be deleted or changed until Zelda could not recognize them.

Digging into the data, we discovered that this behavior was due to a quirk of the diff files that we were using. Often, links were lost when a small number of lines before a link participant was deleted or modified, and a large number of lines were added immediately following. This combination of changes tended to produce a diff result that caused the link participant to be lost. For example, in an instance where the links were incorrectly reported, 12 lines directly before the links were modified and 250 lines were added in between the modified lines and the links. In this case, the diff utility incorrectly reported that the endpoints of our links were modified and changed to the added lines.

Surprisingly, new link locations were reported correctly in a subsequent version of the file. It was possible for these links to reappear, because we were building a link map between a different pair of files.

For example, at the third release after C_1 was created the precision and recall improved, due to 12 lines that were modified in the first release were restored to the original content as when the links were created. Although the added lines were still there, the diff utility now correctly recognized that the lines that are endpoints of our links exist in both version of the file.

Concern Maps with Frequently Changed Code. We saw that links even in concern maps that had frequently changed code, such as C_3, were preserved very well. Despite hundreds of revisions, Zelda was able to attain precision and recall rates of 0.98. The number of changes in the participants of C_3 is summarized in Table 5.

Table 5: Underlying Artifacts of C_3

File Name	No. of Revisions	No. of Links
HistoryModel.java	14	27
jedit_gui.props	357	1
MiscUtilities.java	115	16
SearchAndReplace.java	42	20

Table 5 shows that files with first and second most number of links in C_3 also went through a considerable number of changes. The investigation in to each change shows that these changes touched different parts of the files and even include changes in various method signatures. However, their common characteristic is that most changes involve small amount of lines, modified, inserted or deleted, at each location of change. Due to this characteristic of changes, links in C_3 rarely encounter the same problem as those of C_1 and were retrieved and evolved correctly.

We noticed that the recall rate for all concern maps decreased after release version 4.2 pre 1. The reason for this fall across the board is that the contents of the file CHANGE.txt were completely changed. This file was a participant in all the concern maps except C_3. Previously, the file contained the history for changes made in release 4.1, but in the new version the file contains changes made in release 4.2. The increased in recall of C_4 after this even was coincidental and could be attributed to recovering a couple of links. This small change had a big impact on recall, because C_4 is a small concern map.

7.3 Discussion

It is difficult to directly compare our results with those obtained by Dagenais et al. with ISIS4J [4]. In their tool, participants in concern maps were methods, not lines. As well, their approach used both intensional and extensional representations. However, their study does serve as a useful point of comparison, because they tracked a single complex concern over 34 versions of jEdit, many of which overlap with our study. Initially, they created a concern map with 24 elements and a final concern map with 27 elements. Using only an extensional representation with no explicit support for evolution, they found that only 7 elements remained and were consistent. This corresponds to a precision rate of 0.26. With the addition of ISIS4J, the final concern map had 18 elements, corresponding to a precision rate of 0.67. While it is difficult to say definitively, the performance of Zelda is at least as good as ConcernMapper and ISIS4J together.

We found that in this implementation of Zelda, we lost some links due to the current diff utility we are using cannot recognized that the lines exist in both version of the file when there are changes in lines close to link participants plus a large addition of lines in between. To investigate whether this issue is common among diff utility, we obtained the diff result of the files containing lost links in C1 due to this reason from another diff utility, WinMerge [18]. The diff result from this utility is not in a unified format, but provides the same type of information. We found that analyzing this new diff result allow us to retrieve the previously lost links. This is a promising result, because it shows that these losses were not due to our approach and could easily be remedied. In fact, multiple diff utilities can be used when tracing links to provide sanity check for each other and improve the correctness of retrieved links.

There is a question of scalability of Zelda in dealing with a larger system, which could have millions of links. However, this issue can be easily dealt with by switching to a better database engine. In our system, the links are mainly maintained in the database and retrieved only on demand for a specific concern. Therefore, most links will stay in the

database and the number of links that Zelda retrieves, analyzes, and stores in the memory at a time will be similar to the number of links that Zelda worked with in this experiment. In addition, the link information retrieval speed could also be improved by using database techniques such as indexing. Although we are using MySQL in the current implementation, it is possible to easily change it to another database engine as our connection is made through JDBC, and not depend on any native implementations of MySQL.

7.4 Threats to Validity

Our study provides an evaluation of our approach in evolving extensional representations of concerns when the underlying artifacts change. There are some threats to the validity of our study. First, a threat to the external validity is the characteristic of our subject system and evolution. The ability to generalize the performance of Zelda in evolving link presented in our study depends on how well the characteristic of the code base and the evolution of jEdit could serve as a representative of other software systems. JEdit is a well-known open source software project and the activity level is typical of active, popular projects. The size, frequency of changes, and sequence of changes are representative of open source. Especially, these changes were performed during a long period of time, from 2002 to 2006. In addition, it has been used in various evaluation of concern evolution approaches [4, 12]. However, there are limitations on how applicable our findings are for industrial, closed source software projects.

The next threat to validity is how the concern maps were selected. Select concern maps with participants that change infrequently could bias the results in favor of our tool, thus affecting internal validity. Prior to this study, we were not familiar with jEdit and did not know which files to preferentially include or exclude. We selected our concerns by establishing some criteria and examining the revision control log in sequence until we found five suitable candidates. These concerns seemed to represent a range of sizes, underlying artifacts, and purposes. A beneficial side effect was that all of our concern maps link to files in the top-ten changes list in jEdit, furthering strengthening our results.

8. Future Work

The work in this paper can be extended in a number of ways. While our approach has focused on keeping concern maps up to date while the underlying software artifacts evolve, concerns themselves can evolve as well. The intent behind a concern can shift subtlety or dramatically. It could be a change in connotation (meaning) or denotation (membership in the relation). Our model from Section 3 would need to be modified to include the concept of versioning for concerns as well as software artifacts.

Another drawback of the current concern map evolution algorithm is that it does not handle large-scale changes well. For example, cutting and pasting sections of the code, moving lines between files, or refactoring the code causes links to be lost. It is hoped that the addition of the automatic association mechanism will aid the transfer of link information from one place to another. Another possible option is to deploy existing algorithms in the software evolution area to detect split, merge, and evolution of a file so that we can update the links accordingly [19, 20]. Our approach could also be combined with other tools. As mentioned in Section 2, extensional representations are more flexible and portable, which means that concern maps created using intensional or extensional representations could be evolved using Zelda. One possible usage scenario would involve mining aspects out of source code, a developer checking these aspects manually, and storing these concern maps in Zelda to be evolved over time. Also, our approach could be combined with the algorithms and tools being developed to evolve intensional representations of concerns, such as those found in ISIS4J or INARI. The result would be a single mechanism to evolve both intensional and extensional concerns.

9. CONCLUSION

Extensional representation of concerns provides a flexible means to map concerns to their manifestation in source code. This flexibility is important when varieties of concerns are informal, spontaneous, and scattered in many contexts, which make it difficult to represent them intensionally. In order to make the extensional representation of concern useful in the long-term, concern management tools needs to provide support to overcome the issue of evolving the concern maps as the software evolves. To this end, we have presented an approach to evolving extensional representations of concerns. Our approach has been implemented in Zelda platform, an Eclipse plug-in, for studying evolution of concerns and concern maps.

Zelda allows developers to create concern maps by linking to lines where the concern is manifested in software artifacts. Therefore, our concern map consists of ‘links’ instead of program elements as employed by existing tools. This makes the association mechanism in Zelda highly flexible and allows it to work with any text-based software artifact. By integrating Zelda with a development workbench, the tool is lightweight and easy to use. Concern maps are presented in an editor using markers and in See Soft-style visualization.

Our technique for evolving concern maps takes advantage of a common tool for managing software as it changes over time, the revision control (RC) system. We use the diff results produced by the RC system to keep the location of participants in the concern up to date. By analyzing the diff result between the version of the file where a link to a concern map was added and the version of interest, we can automatically determine the location of participants.

We evaluated the ability of Zelda to accurately maintain links using releases of jEdit, an open-source text editor written in Java (260 KLOC). The result showed that, after 25 versions of concern map creation, the overall precision for the entire set of 419 links across the five concern maps was 0.90 and the recall was 0.73. The average precision and recall per concern map is 0.78 and 0.69 respectively.

References

[1] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, "Aspect-Oriented Programming," in Proceedings European Conference on Object-Oriented Programming, Berlin, Heidelberg, and New York: Springer-Verlag, 1997, pp. 220-242.

[2] A.H. Eden and R. Kazman, "Architecture, Design, Implementation," in ICSE '03: Proceedings of the 25th International Conference on Software Engineering, pp. 149-159, 2003.

[3] M.P. Robillard and G.C. Murphy, "Representing Concerns in Source Code," ACM Trans.Softw.Eng.Methodol., vol. 16, pp. 3, 2007.

[4] B. Dagenais, F.W. Warr and M.P. Robillard, "Inferring Structural Patterns for Concern Traceability in Evolving Software," in ASE '07: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 254-263, 2007.

[5] M.P. Robillard and F. Weigand-Warr, "ConcernMapper: Simple View-Based Separation of Scattered Concerns," in Eclipse '05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, pp. 65-69, 2005.

[6] T. Reinikainen, I. Hammouda, J. Laiho, K. Koskimies and T. Systa, "Software Comprehension through Concern-Based Queries," in ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension, pp. 265-270, 2007.

[7] W. Harrison, H. Ossher, S. Sutton and P. Tarr, "Concern Modeling in the Concern Manipulation Environment," SIGSOFT Softw.Eng.Notes, vol. 30, pp. 1-5, 2005.

[8] F.W. Warr and M.P. Robillard, "Suade: Topology-Based Searches for Software Investigation," in ICSE '07: Proceedings of the 29th International Conference on Software Engineering, pp. 780-783, 2007.

[9] M. Kersten and G.C. Murphy, "Mylar: A Degree-of-Interest Model for IDEs," in AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, pp. 159-168, 2005.

[10] I. Majid and M.P. Robillard, "NaCIN: An Eclipse Plug-in for Program Navigation-Based Concern Inference," in Eclipse '05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, pp. 70-74, 2005.

[11] R. DeLine, M. Czerwinski and G. Robertson, "Easing Program Comprehension by Sharing Navigation Data," in VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), pp. 241-248, 2005.

[12] M.P. Robillard, "Tracking Concerns in Evolving Source Code: An Empirical Study," in ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, pp. 479-482, 2006.

[13] S.M. Sutton Jr. and I. Rouvellou, "Modeling of Software Concerns in Cosmos," in AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development, pp. 127-133, 2002.

[14] E. Soloway, R. Lampert, S. Letovsky, D. Littman and J. Pinto, "Designing Documentation to Compensate for Delocalized Plans," Commun ACM, vol. 31, pp. 1259-1267, 1988.

[15] K. Hrbacek and T. Jech, Introduction to Set Theory, 1999, .

[16] S.G. Eick, J.L. Steffen and E.E. Sumner, "Seesoft-A Tool for Visualizing Line Oriented Software Statistics," IEEE Transactions on Software Engineering, vol. 18, pp. 957, 1992.

[17] jEdit, http://www.jedit.org/2008.

[18] WinMerge, http://winmerge.org/2008.

[19] D. Dig, C. Comertoglu, D. Marinov and R. Johnson, "Automated Detection of Refactorings in Evolving Components," pp. 404-428, 2006.

[20] L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," IEEE Trans.Softw.Eng., vol. 31, pp. 166-181, 2005.